Application for United States Patent

of

David H. Evans, *et al.*

for

Debugger Probe for Object Oriented Programming

CROSS-REFERENCE TO RELATED APPLICATIONS

(CLAIMING BENEFIT UNDER 35 U.S.C. 120)

Not applicable.

FEDERALLY SPONSORED RESEARCH AND DEVELOPMENT STATEMENT

This invention was not developed in conjunction with any Federally sponsored contract.

MICROFICHE APPENDIX

Not applicable.

INCORPORATION BY REFERENCE

United States Patent Number 5,901,315, issued to Jonathan W. Edwards, David H. Evans, and Michael G Vassil, on May 4, 1999, is hereby incorporated by reference in its entirety, including description and diagrams.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates generally to computer software development tools and more particularly to a method and system for debugging a Java application that includes native method dynamic load libraries (e.g., C or C++ code).

Description of the Related Art

Traditional programming methodologies are organized around actions and logic. Programs developed using traditional software design methodologies are usually easily represented by flow diagrams and could be viewed as a procedure of steps which receive input, processes the data, and produces output.

Object Oriented Programming ("OOP"), however, is organized around objects and data. OOP approaches programming from a perspective centralized on providing objects, each object having an interface through which it may be used or by other programs and objects to perform a "method" or set of methods.

One of the earliest object-oriented computer languages was Smalltalk[TM]. More recent and perhaps more widely used object-oriented languages are C++ and Java.

The Java programming language is especially adapted for use in distributed applications, such as on corporate networks and the Internet. Additionally, Java applications have found extensive use on the World Wide Web.

Java developers often "extend" a Java application by linking their own Java code with native method dynamic load libraries ("DLL"). A "native method" DLL is usually a function written in another programming language, such as "C" or "C++", which is not portable across different operating platforms without recompilation.

When the Java application includes linked native method DLL's, a problem arises in debugging the application. In particular, while "tools" for debugging Java code, on the one hand, and tools for debugging C/C++ code, on the other hand, are well-known, the art lacked the ability to simultaneously debug a piece of software that included both types of code.

As a result, some available methods addressed the problem of debugging Java applications having linked native method DLL's by using a "brute force" approach by using distinct debug routines that are run separately. This approach required the Java application with the linked native language DLL's to be executed twice, one for each debug routine, thereby substantially increasing the time necessary to debug the Java application.

The system and method disclosed in US Patent Number 5,901,315 to Edwards, *et al.*, provided a solution to this problem at the time of its availability and in keeping with currently available companion Java development tools. This system and method provided for debugging a target application which comprised Java code having linked native method DLL's which was carried out in a computer having an operating system, a system debug application programming interface ("API"), and a Java virtual machine ("JVM") having a Java debug API.

According to the Edwards system and method, the JVM was first launched under the system debug API, and the Java application was then run under the JVM. Because the JVM that runs the target application itself runs under the system debug API, simultaneous control of the target application via the system debug API and the Java debug API was enabled. Thus, the method allowed the debug of the target application by simultaneously debugging the Java code and the native method dynamic load libraries. Events generated during the debug process were characterized as being initiated from the Java code or the native method DLL's, as the case may be. The preferred embodiment of the Edwards debugger comprised a graphical user interface ("GUI") front end, a debug engine back end, and a "probe" or daemon. The GUI provided an interface through which the user made requests and viewed the status of the application being debugged. The debug engine performed the debugging work by controlling the probe to manipulate the target application through the JVM, and to report events to the debug engine.

The probe was preferably implemented as two distinct processes, a first "daemon" process that performed native method debugging, and a second "probe" process that performed the Java method debugging. The first process also preferably controlled the second process across the system debug API and communicated therewith via a socket connection to facilitate the simultaneous debugging of the Java and C/C++ code comprising the target application.

According to another feature of the Edwards method, the front and back end components of the debugger were supported on a first computer, while the first and

second "processes" were supported on a second computer. The first and second computers could then be connected over a computer network such as a TCP/IP connection to facilitate remote debugging of the target application. The Edwards system and method was implemented in a programming tool product from International Business Machines ("IBM") under the name Interactive Code Analysis Tool ("ICAT").

The current ICAT for Java utilizes a GUI to permit source-level debugging of Java applications on Microsoft's Windows[TM] NT 4.0 and Windows2000. Using ICAT, application developers can set breakpoints, execute and step their applications, and examine the application's stack and variables all with the click of a mouse. If desired, the debugger can be operated remotely; that is, ICAT resides on one system, and the probe resides on the system containing the application. Communication between the two systems is via a TCP/IP connection. The ICAT probe is written in Java, and it interacts with the JVM as needed to control the application being debugged. Essentially, ICAT starts a Windows process, which is the JVM running the probe. ICAT then communicates with the probe via TCP/IP. If ICAT is being run remotely, a daemon is run on the target system which interacts with the probe.

For versions of the JVM before version 1.3, the ICAT probe utilized Sun Microsystems' sun.tools.debug Application Programming Interface ("API") in order to control the application being debugged. The sun.tools.debug API is easy to use, but it is missing a number of functions that the ICAT customarily provided for other debuggers for other programming languages.

Beginning with version 1.3 of the JVM, several new debugger interfaces were provided. Further, support for the sun.tools.debug API was no longer supported. The new Sun product is called the "Java Platform Debugger Architecture" ("JPDA"), and it has three components:

(1) the Java Virtual Machine Debugging Interface ("JVMDI"), which is a low-level native interface which defines what support a JVM must provide in order to allow an application to be debugged;

(2) the Java Debug Wire Protocol (JDWP), which allows communication between a debugger virtual machine and a target virtual machine; and

(3) the Java Debug Interface (JDI), which is a high-level interface, which uses JDWP and JVMDI.

New JDPA features include the ability to exclude classes while stepping, modify Java variables, use deferred breakpoints, create watchpoints, display thread names, attach to running processes, and set breakpoints at arbitrary bytecode addresses. JPDA also allows for capturing output from applications for display by the debugger and permits sending input from the debugger to applications. Further, JPDA is believed to be more stable than the sun.tools.debug API.

There are some known disadvantages to using JPDA. Many functions that the probe performs are more complicated to implement with JPDA, especially exception

filtering. Also, the ICAT function which allows users to load classes is not officially supported in JPDA.

In as much as the existing ICAT does not support JDPA, and whereas JDPA provides improved functionality and stability for Java application program development, testing, and debugging, there is a need in the art for a system and method such as ICAT which supports JDPA and its new functionality.

BRIEF DESCRIPTION OF THE DRAWINGS

The following detailed description when taken in conjunction with the figures presented herein provide a complete disclosure of the invention.

FIGURE 1 shows the arrangement of the components of the debugger system within a single computer system.

FIGURE 2 illustrates the arrangement of the components of the debugger system when debugging is performed from a host system while executing the target application on a remote target system.

FIGURE 3 shows the process of launching an application using the sun.tools.debug API.

FIGURE 4 depicts the process of launching an application by the present invention using JDPA.

FIGURE 5 sets forth the process of handling events using the sun.tools.debug API.

FIGURE 6 depicts the process of handling events by the present invention using JDPA.

FIGURE 7 illustrates the process of stepping through an application using the sun.tools.debug API.

FIGURE 8 depicts the process of stepping through an application by the present invention using JDPA.

FIGURE 9 sets forth the process of setting breakpoints in an application using the sun.tools.debug API.

FIGURE 10 shows the process of setting breakpoints in an application by the present invention using JDPA.

FIGURE 11 sets forth the process of loading classes using the sun.tools.debug API.

FIGURE 12 depicts the process of loading classes by the present invention using JDPA.

FIGURE 13 illustrates the process of providing a console message window for the present invention.

FIGURE 14 shows the use of the Event Request Manager as employed by the present invention.

FIGURE 15 is a block diagram of a representative computer in which the present invention is implemented in whole or in part, including the main components of the debugger of the present invention.

SUMMARY OF THE INVENTION

A method and system are disclosed which provide improved source-level debugging capabilities of an object-oriented application program which may include linked native language dynamic load libraries. The improved debugger is compatible with the Java Platform Debugger Architecture (JDPA), and provides new capabilities such as patching of Java variables and reading and writing strings from and to the application under test and being run by a local or remote Java Virtual Machine.

DETAILED DESCRIPTION OF THE INVENTION

The designers of JPDA intended that most debugger developers, e.g. those developers who design debug tools, to use JDI. According to the preferred embodiment of the present invention, JDI is adopted as the interface for the improved ICAT probe. By modifying the currently-available ICAT to use JPDA, users of the new ICAT have access to the new features immediately and in future ICAT releases. For clarity and distinction, the existing debugger tool will be referred to as "ICAT", and the present invention will be referred to as "ICAT2".

According to the preferred embodiment, the existing ICAT probe design as disclosed in US Patent Number 5,901,315, to Edwards, *et al.*, is used as a starting point, with modifications as disclosed herein in order to migrate the existing ICAT to support JPDA. An advantage to this approach included using existing ICAT capability for instantiating multiple debuggers on multiple platforms. However, it is possible to realize the net design disclosed herein in conjunction with the design disclosed in the Edwards patent as an entirely new program or tool.

Method names and interfaces used herein are consistent with Sun Microsystems' JDPA API version 1.0, which is well known in the art and for which documentation is freely available. Sun Microsystems provides a full reference set for JDPA from their web site or for purchase in print form. Specifically, the "Java[TM] Platform Debugger Architecture Java[TM] Virtual Machine Debug Interface Reference" is hereby incorporated by reference to this disclosure.

As will be described in more detail below, one or more of the processes that comprise the "debugger" of the present invention may be carried out on a computer, or on one or more computers connected via a computer network. Referring to FIGURE 15, a computer for use in realizing the present invention is shown. The software development workstation computer 150 has a processor 151, an operating system 152, an operating system debug application programming interface ("API") (153) and a Java virtual machine (JVM) interpreter (16). The JVM has an associated Java debug API (154) to enable an application program designer to debug Java code.

Thus, for example, the computer (150) used in the present invention is any personal computer or workstation platform, such as an Intel[TM]-, PowerPC[TM]- or RISC[TM]-based computer, and that includes an operating system such as IBM[TM] OS/2[TM], Microsoft Windows[TM] '95/98/2000/NT, UNIX, or IBM AIX[TM]. Such a representative computer runs an Intel Pentium[TM] processor, OS/2 Warp Version 3 operating system, the DosDebug system API and JVM Version 1.3 or greater.

Also shown in FIGURE 15 is a high-level block diagram of the main components of the debugger of the present invention, described in more detail in the following paragraphs.

ICAT2 on a Single System

FIGURE 1 discloses the relationship (10) of the various components of ICAT2 when it is used to debug an application with the debugger running in the same

computer platform as the application. A users graphical user interface (11) is provided to allow the user to view status, data, and to input commands to load processes, set breakpoints, etc. An engine (12) is used to communicated to a daemon DLL (14), preferably by a well-known protocol such as TCP/IP messages (13). The daemon DLL (14) in turn communicates to the Java Virtual Machine (16) which is running the ICAT2 probe (41). Again, communications to the JVM (16) is preferably TCP/IP messages (15). Finally, the ICAT2 probe (41) communicates to a separate instantiation of the JVM (16) running the target application (17).

In this example, both the target application (17) and the ICAT2 debugger (10) are running on the same computer platform, so the communications between the various components of the system are usually internal to the computer itself.

Running ICAT2 on Two Systems

Turning to FIGURE 2, the ICAT2 user interface is run on a local computer, or "host system", (20), while the target application (17) is run by the JVM (16) on a remote "target system" (21). This is a common arrangement found during Java software development, where a number of potential target systems may be available over a computer network for a designer to use for debugging an application. The user then establishes communications with the target system using the user's own workstation as a host system. The choice of TCP/IP for communications within the single-system arrangement lends itself to dividing the components of the system between a host system (20) and a target system (21). In this case, the TCP/IP

communications (13) between the engine (12) running on the host system (20) to the daemon (14) running on the target system (21) is carried preferably over a computer network such as a Local Area Network ("LAN") or even the Internet.

Connecting to the JVM

First, the ICAT2 probe must establish a connection to the JVM. With the sun.tools.debug API, this was a relatively straightforward process, as shown in FIGURE 3. A Remote Debugger ("RD") object (31) was instantiated (32) by the probe (18). The parameters to the RD constructor included the name of the application to be debugged (and any parameters to be passed to the application) and a pointer to the probe, which told RD that the probe would provide the required event handlers (breakpointEvent, exceptionEvent, etc.) (34).

Turning to FIGURE 4, the process of connecting to the JVM is a bit more complicated with JPDA, but JPDA supports multiple types of connectors, which facilitate attaching and listening, in addition to launching. ICAT2 supports launching applications through JDPA's LaunchingConnector (LC) (44), as shown in FIGURE 4.

JPDA provides a Bootstrap object (42) that allows access to the JDI interfaces. The Bootstrap object (42) is used to obtain the VirtualMachineManager (43). The VirtualMachineManager (43) provides a list of connectors of various types. The present invention, ICAT2, iterates through the list of connectors, and searches for "com.sun.jdi.CommandLineLaunch", which is stored away as a Connector object.

Launching the Application

In the older ICAT with the sun.tools.debug API, the launching process was accomplished when RD was instantiated. However, according to the present invention ICAT2 with JPDA (40), the application is launched when the probe function "launch_then_attach" (46) is called. There is a set of arguments associated with each connector. The "main" argument is set to the name of the application, and the "suspend" argument is set to true. Then, ICAT2 invokes the launch method on the "LC' object. The parameters given to the method are connector arguments. The launch method returns a VirtualMachine (VM) object (45) to the probe (41).

The VM object (45) is similar to the RD object in the sun.tools.debug API. The VM object (45) mirrors the state of the JVM and provides the objects and methods needed for controlling the application under debug, as did the previous RD object.

Next, the probe obtains the EventRequestManager ("ERM") from the VM object (45), as shown in FIGURE 14. The event handling code in the ICAT2 probe (41) makes extensive use of the ERM (80). First, an exception request (140) is created to tell the JVM to suspend the application in response to any exception that occurs. This is the default. In ICAT2, the user can later select whether to catch or ignore specific exceptions that occur within "try" blocks. Next, the modified ICAT2 creates a request (141) to be notified each time a class is loaded. In JPDA, notifications of class loads are called ClassPrepareEvents. ICAT2 also request to be informed of thread death events.

For each of the request objects created above, ICAT2 must indicate that it requires the JVM to suspend the application at the time an event occurs. This is done by calling the setSuspendPolicy method on each object and passing the parameter SUSPEND_ALL. This indicates that all JVM threads are to be suspended when an event occurs. Finally, the request objects are enabled by invoking the "enable" method.

The ICAT2 probe then enters a loop in which events are received, shown in FIGURE 6 and described in more detail later. These events are processed until the class prepare event is received for the application that was launched. At this point, the ICAT2 probe (41) reads in a list of the currently loaded classes (also called "modules"). Then, a thread is spawned which enters an event retrieval loop that runs continuously as long as the probe is active. This completes the processing for the launch_then_attach function.

Obtaining Stack Information

The ICAT2 probe relies heavily on the facilities for obtaining information about the "call stack". This is handled by the ICAT2 probe similarly to the ICAT probe process. With each API, the thread object is used to obtain the data. In the sun.tools.debug API this object is the RemoteThread (RT), in JPDA it is ThreadReference (TR).

Handling Events

The JVM notifies the debugger of asynchronous occurrences by using events. The probe MUST provide event handlers for certain events, even if it chooses not to take any action. This is true for both the sun.tools.debug API and JPDA.

As shown in FIGURE 5 with the sun.tools.debug API, in ICAT the only events reported were the ones for which event handlers were provided. To generate other events, the probe had to provide code to check constantly, or "poll", for new modules and threads. The ICAT probe then sent, for example, a "module loaded" event to ICAT.

With JPDA, there are more event handlers that must be provided by the ICAT2 probe, as disclosed in FIGURE 6. Also, a debugger designer can optionally request to be notified of a number of additional events (for example, see "Launching the Application" above). The VM object (45) has an EventQueue (46) which must be polled (60) to see these requested events, which are objects themselves. In the ICAT2 probe (41), events (600) are taken off the event queue (69) and checked to determine the event type. A call is then made to the proper event handler (61 through 67) for each event.

The processing for an event is notably different between the two APIs, sun.tools.debug and JPDA. With the sun.tools.debug API, the exception handlers were called with an RT object provided. The information about the event was obtained from the RT. However, with JPDA, the information is obtained from the event object.

Further, JPDA adds several new event handlers. The ICAT2 probe provides a function for each of these, although many of these functions may be empty.

The stepEvent is new with JPDA. With the sun.tools.debug API, the completion of a step or a breakpoint hit was considered a "breakpointEvent". The probe code had to sort out the situation and determine what had happened. However with JPDA, there are separate handlers for stepEvent and breakpointEvent, and therefore separate event handlers (61 and 62) are provided by the ICAT2 probe (41).

With JPDA, the JVM generates an event whenever a class is loaded. This is called a ClassPrepareEvent. This new JPDA capability is a major improvement over the sun.tools.debug API, through which it was necessary to poll the JVM periodically for a list of loaded classes to determine if any new classes had been loaded. Thus, a ClassPrepare event handler (63) is provided in the ICAT2 probe (41).

JPDA provides other new events that are not used in the preferred embodiment, including: VM start event, field watch event, VM interrupted event, method entry event and method exit event. The ICAT2 probe (41), however, provides a handler for each of these events, though the handler doesn't necessarily act on the events.

Exception event handling is similar between JPDA and the sun.tools.debug API. As mentioned before, the information about the exception comes to the probe in an ExceptionEvent object using JPDA, rather than from the RT object in under the sun.tools.debug API. An exception event handler (64) is provided in the ICAT2 probe (41) to handle receipt of an ExceptionEvent.

Thread death events must be handled with the sun.tools.debug API but are optional with JPDA. The sun.tools.debug API has a quit event which is analogous to the VM death event with JPDA. The architecture of ICAT2 requires notification of thread death events, so the ICAT2 probe (41) requests these events from the VM object (45). A VM death event handler (65) is provided in the ICAT2 probe (41) to handle receipt of an VMDeathEvent.

Further, thread start event and thread death event handlers (66 and 67) are provided in the ICAT2 probe (41).

Stepping

Issuing a step in the sun.tools.debug API was fairly straightforward, as shown in FIGURE 7. The thread object (70) had a "step" member function (71). The parameter to this function was a Boolean which indicated whether the step was to be a line step or an instruction step.

With JPDA, the ICAT2 probe (41) first obtains the ERM (80) from the VM object (45), as shown in FIGURE 8. From the ERM (80) a list of any outstanding step requests is obtained for the thread of interest. ICAT2 may delete any outstanding requests found. The ERM (80) has a method called "createStepRequest" (81). The parameters to this method indicate the type of step desired. The first parameter is the TR of the thread to be stepped, and the second is the size of the step. There are two sizes, "min" (minimum - which is generally an instruction step), and "line". The third parameter is depth, which may have the values "into", "over" and "out".

A count filter is added (83) to the request (82) next. The count filter is preferably set to 1, which gives the ICAT2 probe (41) a single notification when the step is completed. Finally, the request is enabled (84) and the JVM is resumed.

Obtaining Memory Usage Information

For ICAT2's memory usage window, few changes from ICAT were required. The information about free memory and total memory was available to ICAT from the RD with the sun.tools.debug API, which is now available to ICAT2 from an object called "Runtime" with JPDA. As such, ICAT2 obtains memory usage information from the JDPA Runtime object.

Obtaining Thread Names

JPDA allows ICAT2 to obtain the name of any thread, whereas the sun.tools.debug API RD did not support this. The TR object has a method that returns the name, which is employed by ICAT2 to obtain the name of any thread.

Filtering Exceptions

ICAT2 allows the user to choose whether or not to have the debugger report exceptions that occur within "try" blocks. When an exception occurs outside a "try" block, execution always stops and the exception is reported to the user.

With the sun.tools.debug API, catching or ignoring exceptions in ICAT was a relatively simple process. The RemoteClass (RC) object had both an

"ignoreExceptions" and a "catchExceptions" method. In order to obtain an exception object, the "findClass" method on RD was called. This loaded the class if it is not already loaded.

The exception filtering process for ICAT2 with JDPA is more complicated than with sun.tools.debug. If the exception class of interest is not loaded, it must be loaded according to a process described below. If a request object for the exception already exists, ICAT2 disables and deletes it so that a new one can be created. When ICAT2 calls "createExceptionRequest", one of the parameters to the method is a Boolean which indicates whether caught exceptions (exceptions within a "try" block) are to be reported or ignored. The newly created "ExceptionRequest" object is enabled after setting the suspend policy.

Loading Classes

There are several circumstances in which a debugger such as ICAT or ICAT2 must force the loading of a class. ICAT provided a function with which the user could request the forced loading of a class. As shown in FIGURE 11, this was relatively uncomplicated with the sun.tools.debug API since the "findClass" method (92) automatically loads a class if it is not already loaded.

However, class loading is a multi-step process with JPDA, as shown in FIGURE 12. ICAT2 first must obtain a reference to java.lang.ClassLoader (120). ICAT2 then has to obtain a reference to the method "getSystemClassLoader" (121), which is then invoked. In order to do so, ICAT2 must have a reference to an

application thread. Unfortunately, this does not work for any thread other than the main thread, so a reference to the main thread (122) is obtained by the ICAT2 probe. Since the main application thread can terminate before the application terminates, it is possible that the user will see a situation in which it is impossible to force the loading of a class.

Obtaining Thread Status

JPDA has the ability to obtain the status of any application thread, such as "running", "sleeping", "waiting", "not started", etc. The TR object has a member function called "status" which returns this information when invoked by the ICAT2 probe.

Providing and Displaying a Console Message Window

The move to JPDA allows ICAT2 to provide a console window, as shown in FIGURE 13. This permits users to see output printed by the application, and to send data to the application. In order to allow writing data to the application, the ICAT2 probe (41) first obtains the "Process" object (130) from the VM object (45). The "OutputStream" (131) is obtained from the "Process" object (130). The OutputStream is used to instantiate a "BufferedWriter" (133). The "write" method (136) on the "BufferedWriter" object (133) is called in order to write data to the application from the console message window.

For obtaining data that is printed by the application and presenting it on the console message window, a "BufferedReader" object (134) is created in a manner similar to that used to create a BufferedWriter, including obtaining the Process object (130) from the VM object (45), obtaining the InputStream object (132) from the Process object (130), instantiating a BufferedReader object (134), and using the "read" method (135) to retrieve data from the application.

Setting Breakpoints

Turning to FIGURE 9, the process employed by the ICAT probe (18) with the sun.tools.debug API to set a breakpoint is fairly simple, which involved calling the findClassmethod (92) of the RD (31) and the "setBreakpointLine" method (91) on the class where the breakpoint was to be placed. The line number provided was the source line. There was no provision for setting breakpoints at an arbitrary bytecode address.

The ICAT2 probe (41) with JPDA first obtains a list of the loaded classes (102), as shown in FIGURE 10. If the class where the breakpoint is to be placed is not loaded, the setting of the breakpoint is deferred by adding it to deferred breakpoint table (101) to be installed later. If the class is loaded, the line number information is obtained from the class object (103). Then the breakpoint request (106) is instantiated (107) by the ERM (80), its suspend policy is set (108), and the request is enabled (109).

Clearing Breakpoints

Clearing a breakpoint was similarly straight forward with the sun.tools.debug API for ICAT. The method that was used was "clearBreakpointLine".

For ICAT2 with JDPA, a list of the active BreakpointRequests must first be obtained. If ICAT2 finds one that matches the user's requested breakpoint deletion, that BreakpointRequest object is deleted using a JDPA API command.

Halting an Application

In order to halt the application, ICAT2 obtains a list of the current application threads by calling the VM's "allThreads" method. ICAT2 then iterates through this list and invokes the JDPA API "suspend" method on each thread that is not already suspended.

Requesting Thread Information

When ICAT2 obtains the list of threads from the VM object (45), it checks to make sure that the thread group is the main group, which is the group of application threads. Any threads that are in the main group are added to the list that is returned.

Resuming Execution

The "resume" command to the probe is a request to restart execution of the application. Resuming is one area that may be simpler with JPDA than with the sun.tools.debug API.

ICAT invoked one of three methods through the sun.tools.debug API: the "run" or "cont" method on the RD object, or the "resume" method on any particular RemoteThread object.

ICAT2 using the JPDA API reviews the current list of step requests and deletes any that are currently active. Then, ICAT2 invokes the "resume" method on the JVM object.

Excluding Classes

During migration from ICAT with sun.tools.debug to ICAT2 with JPDA, a new feature was brought to ICAT2: the ability to exclude classes when stepping. This allows users to prevent ICAT2 from stopping in certain classes when stepping. It is possible to exclude entire sets of classes by use of an asterisk. For example, to exclude all classes beginning with "abc.def", the user sets the environment variable CAT_CLASS_EXCLUDE=abc.def.*. This would exclude abc.def.ghi, abc.def.jkl, etc. The ICAT2 probe excludes classes by invoking the "addClassExclusionFilter" method on the step request object.

Evaluating Expressions

For expression evaluation, no major changes were needed to the probe to migrate from the sun.tools.debug API to JPDA. A line-by-line replacement of RD objects and methods with VM objects and methods is sufficient. However, the ICAT2 user is now able to modify Java variables.

Initializing Objects

The "ExpressionEvaluator" and "ExpressionServices" constructors are initialized with a VM object in place of an RD object.

Accessing and Querying of Debuggee Classes

The VM contains methods (classesByName and allClasses) for returning ReferenceType objects in the debuggee. The ReferenceType object contains the necessary methods for accessing information about the class under test such as class methods, fields, types, values, static, final, etc.

Adding New Function

Modifying Symbols. JPDA provides "setValue" and "mirrorOf" utilities. SetValue performs modification of static and instance fields, strings, array elements, and Booleans. MirrorOf returns an access reference to the type being modified. A new method, update_value, was added to the probe which utilizes both of these utilities to change dynamically and return new field values to the user during a debug session.

Evaluating Strings. Although ICAT provided string evaluation, ICAT2 with JPDA includes a StringReference interface, which extends ObjectReference. This means that ICAT2 can treat the string as a class, and therefore provide the user with much more information pertaining to the string. The RmtStringValue class was

subsequently eliminated, with the new string evaluation code included in RmtObjectValue class.